Guide to the BOHOS Operating System Project

Version 0.3

Renzo Davoli


January 26, 2008
1. Introduction

The BOHOS Operating System (OS) described below is inspired to the previous experiences of Kaya OS[1], AMIKE OS[2], AMIKaya OS[8] and Mikonos[9] projects. All of them are descending (not directly) from the THE OS[3] outlined by E. Dijkstra. In his papers he described an OS divided into six layers. Each layer $i$ provides an abstraction layer to the $i+1$ layer. Kaya derives from the past experiences of the TINA OS and MPS [6], a rework of the HOCA OS and CHIP [4,5]. The AMIKE and AMIKaya specifications introduced microkernel operating systems, based on the message passing facility.

The OS we are going to describe is not complete as Dijkstra's one.

- Level 0: μMPS hardware, described in the μMPS Principles of Operation)
- Level 1: services provided in ROM (fully described in μMPS Principles of Operation):
  - processor state save/load
  - ROM-TLB-Refill handler
  - LDST, FORK, PANIC, HALT
- Level 2: the Queue/Data Structure Managers (Phase 1A – described in Chapter 2). Based on the key operating system concept that active entities at one layer are just data structures at lower layers, this layer support management of queues of ThreadBLK structures
- Level 3: the Nucleus (Phase 2 – described in Chapters 3). This level implements the thread scheduling, interrupt handling, semaphores.

The Bohos main goal is to design a source code that implements all these levels. Further levels could be developed:

- Level 4: the Support Level. The 3rd level is extended to a system that can support multiple user-level cooperating threads that can request I/O and which run on their own virtual address space. Furthermore, this level adds user-level synchronization, message passing and a thread sleep/delay facility
- Level 5: the Network Level. This level implements a minimal TCP/IP stack to get provide access with existing virtual Ethernet devices to a real network, through VDE[7]
- Level 6: the File System. This level implements the abstraction of a flat file system with primitives to create, rename, delete, open, close and modify files
- Level 7: the Interactive Shell

2. Phase 1 – Level 2: The Queue/Data Structure Managers

Level 2 of BOHOS instantiates the key operating system concept that active entities at one layer are just data structures at lower layers. In this case, the active entities at a higher level are threads, and what represent them at this level are thread control blocks (ThreadBLKs).

```c
typedef unsigned int status_t;  /* thread status */
typedef struct tcb_t {
    struct tcb_t *t_next_l, *t_next_r; /* pointer to next entries in the thread queue/priority queue */
    int priority;
    state_t proc_state;            /* processor state */
    int *t_semAdd;                /* semaphore address (in case of blocked thread) */
    /* other fields will be added during phase2 development */
} tcb_t;
```
The Thread Queue Manager will implement functions to provide these services:
- Allocation/de-allocation of single ThreadBLK elements
- Maintenance of ThreadBLK queues/priority queues
- Maintenance of a hash table of active semaphore descriptors, each of which supports a queue of ThreadBLK

### 2.1 ThreadBLK Allocation/De-allocation

One may assume that BOHOS supports no more than MAXTHREADS concurrent threads; this parameters is defined in the file const.h. Thus, this level needs a number of MAXTHREADS Thread Control Blocks (ThreadBLKs) to allocate from and de-allocate to. Entity allocation and de-allocation will be provided by these procedures.

```c
void initTcb();
```

Initialize the data structures needed for Thread control block allocation/deallocation. It sets up the list of available unused ThreadBLK. This function will be called once before any other operation on ThreadBLKs.

```c
tcb_t * allocTcb(void);
```

Returns NULL if there are no available unused ThreadBLK elements left. Otherwise it extracts an element from the list of unused ThreadBLK and returns a pointer to it. All fields of the returned ThreadBLK must be reset to default values (i.e. NULL and/or 0). ThreadBLKs get reused, so it is important that no previous value remains in a ThreadBLK when it gets reallocated.

```c
void freeTcb(tcb_t *t);
```

This functions frees the ThreadBLK: it inserts the ThreadBLK pointed by t into the list of unused available ThreadBLK.

Since BOHOS will run without any dynamic memory allocation/deallocation facility, runtime resource request is not allowed. The best way to solve this problem is to have a initial allocation of all the required space, whose lifetime would be exactly the same of the kernel’s one. This can be done during initialization of data structures, using a static allocation of one array for the MAXTHREADS ThreadBLKs.

### 2.2 Thread Queue Maintenance

The functions below do not manipulate a particular thread queue or set of queues. Instead they are generic queue manipulation methods; one of the parameters is a pointer to the thread queue upon which the indicated operation is to be performed. Queues to be manipulated are circular, singly linked and tail pointed lists or priority queues.
To provide support to thread queues, the following externally visible functions should be implemented:

```c
void mkEmptyThreadQ(void); // Used to initialize a variable to be tail pointer to a thread queue; returns a pointer to the tail of an empty thread queue, i.e. NULL

int emptyThreadQ(tcb_t *tp); // Returns TRUE if the queue whose tail is pointed to by tp is empty, FALSE otherwise.

void insertBackThreadQ(tcb_t **tp, tcb_t *t_ptr); // Insert the ThreadBLK pointed to by t_ptr at the back of the thread queue whose tail-pointer is pointed to by tp; note the double indirection through tp to allow for the possible updating of the tail pointer as well

void insertFrontThreadQ(tcb_t **tp, tcb_t *t_ptr); // Insert the ThreadBLK pointed to by t_ptr at the front of the thread queue whose tail-pointer is pointed to by tp; note the double indirection through tp to allow for the possible updating of the tail pointer as well

void removeThreadQ(tcb_t **tp); // Remove the first (i.e. head) element from the thread queue whose tail-pointer is pointed to by tp. Return NULL if the thread queue was initially empty; otherwise return the pointer to the removed element. Update the thread queue’s tail pointer if necessary

void outThreadQ(tcb_t **tp, tcb_t *t_ptr); // Remove the ThreadBLK pointed to by t_ptr from the queue whose tail-pointer is pointed to by tp. Update the queue’s tail pointer if necessary. If the desired entry is not in the queue (an error condition), return NULL; otherwise, return t_ptr. Note: t_ptr can point to any element of the queue

tcb_t * headThreadQ(tcb_t *tp); // Return a pointer to the first ThreadBLK from the queue whose tail is pointed to by tp. Do not remove the ThreadBLK from the queue. Return NULL if the queue is empty.
```
2.3 Priority Queue Management

To provide support to thread queues, the following externally visible functions should be implemented:

```c
#include "thread.h"

tcb_t * mkEmptyThreadPQ(void);

  Used to initialize a variable to be  pointer to a thread priority queue; returns a pointer to the
tail of an empty thread queue, i.e. NULL

int emptyThreadPQ(tcb_t *tp);

  Returns TRUE if the priority queue whose root is pointed to by tp is empty, FALSE otherwise.

void insertThreadPQ(tcb_t **tp, tcb_t *t_ptr);

  Insert the ThreadBLK pointed to by t_ptr in the priority queue whose head is pointed to by tp,
the priority value is stored in the structure pointed to by t_ptr; note the double indirection
through tp to allow for the possible updating of the tail pointer as well.

tcb_t * removeThreadPQ(tcb_t **tp);

  Remove the root element from the prority queue whose head is pointed to by tp. Return
NULL if the thread prority queue was initially empty; otherwise return the pointer to the
removed element. Update the priority queue’s structure.

tcb_t * outThreadPQ(tcb_t **tp, tcb_t *t_ptr);

  Remove the ThreadBLK pointed to by t_ptr from the priority queue whose root is pointed to
by tp. Update the priority queue structure. If the desired entry is not in the priority queue (an
error condition), return NULL; otherwise, return t_ptr. Note: t_ptr can point to any element of
the queue

tcb_t * headThreadPQ(tcb_t *tp);

  Return a pointer to the first ThreadBLK from the priority queue whose root is pointed to by tp.
Do not remove the ThreadBLK from the priority queue. Return NULL if the queue is empty
```
A semaphore is an important operating system concept which is needed for Phase 2/Level 3. While understanding semaphores is not needed for this level, this level nevertheless implements an important data structure/abstraction which supports Kaya's implementation of semaphores.

For the purposes of this level it is sufficient to think of a semaphore as an integer. Associated with this integer is its address (semaphores, like all integers, have a physical address) and a process queue. A semaphore is active if there is at least one TsKBlk on the process queue associated it. (i.e. The process queue is not empty: emptyThreadQ(s_threadq) is FALSE.)

The following implementation is suggested: Maintain a hash table of NULL-terminated single linearly linked lists (to manage hash value collisions, using the s_next field) of semaphore descriptors. **Semhash** is the hash table array (struct semd_t), the s_semAdd field is the hash key.

The size if the hash table is provided by the constant ASHSIZE.

```c
/* semaphore descriptor type */
typedef struct semd_t {
    struct semd_t *s_next; /* next element on this ASH entry
    collision list*/
    int *s_semAdd; /* pointer to the semaphore*/
    tcb_t *s_threadQ; /* tail pointer to a thread queue */
} semd_t;
```

Maintain a list of semaphore descriptors, the semdFree list, to hold the unused semaphore descriptors. This list, whose head is pointed to by the variable semdFree h, is kept, like the pcbFree list, as a NULL-terminated single linearly linked list (using the s_next field).

The semaphore descriptors themselves should be declared, like the ThreadBlk's, as a static array of size MAXPROC of type semd_t.

To support the ASH there should be the following externally visible functions:

```c
int insertBlocked(int *semAdd, tcb_t *p)
```
Insert the ProcBlk pointed to by \( p \) at the tail of the process queue associated with the semaphore whose physical address is \( \text{semAdd} \) and set the semaphore address of \( p \) to \( \text{semAdd} \). If the semaphore is currently not active (i.e. there is no descriptor for it in the ASH), allocate a new descriptor from the \( \text{semdFree} \) list, insert it in the ASH (at the appropriate position), initialize all of the fields (i.e. set \( s.\text{semAdd} \) to \( \text{semAdd} \), and \( s.\text{procq} \) to \( \text{mkEmptyProcQ()} \), and proceed as above. If a new semaphore descriptor needs to be allocated and the \( \text{semdFree} \) list is empty, return TRUE. In all other cases return FALSE.

\[
\text{tcb_t *removeBlocked(int *semAdd)}
\]

Search the ASH for a descriptor of this semaphore. If none is found, return NULL; otherwise, remove the first (i.e. head) ProcBlk from the process queue of the found semaphore descriptor and return a pointer to it. If the process queue for this semaphore becomes empty (\( \text{emptyProcQ(s.procq)} \) is TRUE), remove the semaphore descriptor from the ASH and return it to the \( \text{semdFree} \) list.

\[
\text{tcb_t *outBlocked(tcb_t *p)}
\]

Remove the ProcBlk pointed to by \( p \) from the process queue associated with \( p \)'s semaphore (\( p.\text{semAdd} \)) on the ASH. If ProcBlk pointed to by \( p \) does not appear in the process queue associated with \( p \)'s semaphore, which is an error condition, return NULL; otherwise, return \( p \).

\[
\text{tcb_t *headBlocked(int *semAdd)}
\]

Return a pointer to the ProcBlk that is at the head of the process queue associated with the semaphore \( \text{semAdd} \). Return NULL if \( \text{semAdd} \) is not found on the ASH or if the process queue associated with \( \text{semAdd} \) is empty.

\[
\text{initASH()}
\]

Initialize the \( \text{semdFree} \) list to contain all the elements of the array

\[
\text{static semd_t semdTable[MAXPROC]}
\]

This method will be only called once during data structure initialization.

### 2.5 Nuts and Bolts

There isn’t just one way to implement the functionality of this level. Regarding optimization, efficiency may be improved by introducing doubly-linked queues, adding eg. \( \text{t.prev} \) to ThreadBLKs. Each module should export its public interface using a .e file. As with any non-trivial system, you are strongly encouraged to use the make program to maintain your code. Initial structure initialization can be accomplished by static allocation of three data structure, an array for ThreadBLKs, a one word bitmap for ThreadBLK allocation and an array or an hashtable to map TIDs to ThreadBLK pointers. Eg.:

\[
\text{HIDDEN tcb_t tcbTable[MAXTHREADS];}
\]

### 2.5 Testing

There is a provided test file, \( \text{p1test.c} \) that will “exercise” your code. You CANNOT modify \( \text{p1test.c} \). You should compile the source files separately using the commands eg.:

\[
\text{mipsel-linux-gcc -ansi -pedantic -Wall -c tcb.c}
\]

\[
\text{mipsel-linux-gcc -ansi -pedantic -Wall -c ash.c}
\]

\[
\text{mipsel-linux-gcc -ansi -pedantic -Wall -c p1test.c}
\]

The object files should then be linked together using the command:

\[
\text{mipsel-linux-ld -T $SUPDIR/elf32ltsmip.h.umpscore.x $SUPDIR/crtso.o $SUPDIR/libumps.o tcb.o tid.o p1test.o -o kernel}
\]

Where $SUPDIR must be replaced with the path to the \( \mu \)MPS support directory and
elf32ltsmip.h, umpscore.x,  crts0.o and libumps.o are part of the µMPS distribution.

If one is working on a big-endian machine one should modify the above commands appropriately; substitute mips- for mipsel- and elf32btsmip.h.umpscore.x for elf32ltsmip.h.umpscore.x.

The linker produces a file in the ELF object file format which needs to be converted prior to its use with µMPS. This is done with the command:

```
umps-elf2umps -k kernel
```

which produces the file kernel.core.umps

Finally, your code can be tested by launching µMPS. Entering:

```
umps
```

without any parameters loads the file kernel.core.umps by default.

Hopefully the above will illustrate the benefits for using a Makefile to automate the compiling, linking, and converting of a collection of source files into a µMPS executable file.

The test program reports on its progress by writing messages to TERMINAL0.

These messages are also added to one of two memory buffers; errbuf for error messages and okbuf for all other messages. At the conclusion of the test program, either successful or unsuccessful, µMPS will display a final message and then enter an infinite loop.

The final message will either be SYSTEM HALTED for successful termination, or KERNEL PANIC for unsuccessful termination.

**Credits**

The author would like to thank: Claudio Sacerdoti Coen, Michael Goldweber, Mauro Morsiani and Enrico Cataldi for the great work made with µMPS, Kaya, AMIKE, AMIKAYA, MIKONOS [1,2,8].
References


2. Mauro Morsiani, AMIKE OS specifications.


